RAMRAO ADIK INSTITUTE OF TECHNOLOGY,
NERUL

Lab Manual:

Object Oriented Analysis & Design

Department of Computer/IT Engg.

List of Experiments

Name of Faculty: **Vimla Jethani** **Reshma Gulwani**	Term: **Jan 2008 – May 2008**
Subject: **OOAD**	Lab Session: **3hrs/week**
Year & Semester: **T.E Sem VI**	No. of Lectures per week: **3**
Branch : **Computers** **Information Technology**	No. of Weeks: **15**

SR.No.	Title
1	Study of UML tools
2	Problem Definition
3	Class Diagram
4	Use case diagram
5	Sequence Diagram
6	Collaboration Diagram
7	Activity Diagram
8	State Chart Diagram
9	Component Diagram
10	Deployment Diagram
11	Test cases

Experiment No 1

Aim: To study various UML tools.

Theory:

The Unified Modelling Language Tools are classified by their proprietary or non-proprietary status.

Non-Proprietary UML Tools

Eclipse: with Eclipse Modeling Framework (EMF) and UML 2.0 (meta model without GUI) projects.

ArgoUML: argouml.tigris.org a Java-based open source free UML modelling tool, closely follows the UML standard, BSD license.

ATL: a tool which can transform UML models into other models. Available from the Eclipse GMT project (Generative Modeling Tools).

StarUML: an open-source UML/MDA platform for Microsoft Windows, licensed under a modified version of GNU GPL, mostly written in Delphi

Proprietary UML Tools

Potential users can freely download versions of most of the following tools; such versions usually impose limits in capability and/or by a time-period.

Microsoft Visio: a diagramming tool that also supports UML

Rational Rose: by Rational Software (sold to IBM in 2003); supports UML 1.x.

Rational Rose XDE: an "eXtended Development Environment" in the tradition of Rational Rose; supports UML 1.x

Rational Software Architect: Eclipse-based UML 2.0 tool by the Rational Division of IBM

SmartDraw: UML-diagram tool for Microsoft Windows

Study of Rational Rose & StarUML

Rational Rose

Rose facilitates object-oriented analysis and design, better known as OOAD. In fact, Rose is an acronym for Rational Object Oriented Software Engineering. The great thing about Rose is that it allows analysts, engineers, writers, and project managers to create, view, manipulate, and modify elements in a Unified Modeling Language (UML) across the entire enterprise, using one tool and one language. The tool's true value is that it exposes software development problems early on in the development life cycle, helping you manage everything from straightforward projects to more complex software solutions. Basically, Rose supports use-case-driven object modeling.

Rose includes features that simplify the software development process:

- UML modeling
- Multilanguage development
- Component-based development
- Internet Web publisher
- Basic report generator
- Database schema generator

Rose utilizes diagrams as views of the information in a model. Once developed, Rose automatically maintains consistency between the diagram and its specifications. The following key diagrams are used:

- **Use case diagrams:** Analysts and developers use these to capture user requirements by graphically depicting how the system works. During the design phase of the project, Rose allows you to actually specify the system behavior (what Rose calls use cases). The use case diagram therefore graphically represents the system boundary. Typically, a use case diagram consists of (1) actors or things outside the system, (2) use cases, and (3) relationships between actors.
- **Class diagrams:** Rose uses class diagrams to graphically describe generic descriptions of the system you're going to build. Class diagrams contain icons that represent classes and interfaces and their relationships to one another.
- **Statechart and Activity diagrams**: Rose allows users to use statechart diagrams (which are state-driven) to model the dynamic behavior of individual classes or objects. Statechart diagrams are very similar to activity diagrams (which are activity-driven). Basically, these diagrams show you (1) the sequence of states that an object will go through, (2) the events that cause a transition from one activity to another, and (3) any actions that result from the state or activity change.
- **Interaction diagrams:** Rose uses interaction diagrams as a collective name for collaboration and sequence diagrams, which, in essence, graphically represent interactions. Collaboration diagrams show how objects are associated with each other, whereas sequence diagrams show time-based interactions between objects.
- **Component diagrams:** Rose uses component diagrams to clearly reflect the physical dependency relationships between components (i.e., main program, subprogram, packages, and tasks) and their arrangement in a graphical manner.

- **Deployment diagrams**: Using the deployment diagram, Rose allows users to graphically show the connections between processors, devices, and connections.

StarUML

StarUML is an open source project to develop fast, flexible, extensible, featureful, and freely-available UML/MDA platform running on Win32 platform. The goal of the StarUML project is to build a software modeling tool and also platform that is a compelling replacement of commercial UML tools such as Rational Rose, Together and so on.

- **UML 2.0**: UML is continuously expanding standard managed by OMG(Object Management Group). Recently, UML 2.0 is released and StarUML support UML 2.0 and will support lastest UML standard.

- **MDA (Model Driven Architecture)**: MDA is a new technology introduced by OMG. To get advantages of MDA, software modeling tool should support many customization variables. StarUML is designed to support MDA and provides many customization variables like as UML profile, Approach, Model Framework, NX(notation extension), MDA code and document template and so on. They will help you fitting tool into your organizational cultures, processes, and projects.

- **Plug-in Architecture**: Many users require more and more functionalities to software modeling tools. To meet the requirements, the tool must have well-defined plug-in platform. StarUML provides simple and powerful plug-in architecture so anyone can develop plug-in modules in COM-compatible languages (C++, Delphi, C#, VB, ...)

- **Usability**: Usability is most important issue in software development. StarUML is implemented to provide many user-friend features such as Quick dialog, Keyboard manipulation, Diagram overview, etc.

StarUML is mostly written in Delphi. However, StarUML is *multi-lingual project* and not tied to specific programming language, so any programming languages can be used to develop StarUML. (for example, C/C++, Java, Visual Basic, Delphi, JScript, VBScript, C#, VB.NET, ...)

Features

UML 2.0 Diagrams

Use Case Diagram

Class Diagram

Sequence Diagram

Collaboration Diagram

Statechart Diagram

Activity Diagram

Component Diagram

Deployment Diagram

Composite Structure Diagram (UML 2.0)

Conclusion: Thus various software modeling tools can be used by analysts, engineers, writers, and project managers that exposes software development problems early on in the development life cycle, helping to manage everything from straightforward projects to more complex software solutions.

Experiment No 3

Aim: To draw Class diagram.

Theory: Object diagram provide a formal graphic notation for modeling object, classes and their relationship to one another. Object diagrams are useful for both abstract modeling and for designing actual programs. There are two types of object diagram: class diagrams and instance diagrams.

Class diagram is a schema, pattern or template for describing many possible instances of data. A class diagram describes object classes.

An instance diagram describes how a particular set of objects relates to each other. An instance diagram describes object instances. Instance diagram are useful for documenting test cases (especially scenarios) and discussing example. A given class diagram describes infinite set of instance diagrams

Need of Class Diagram:

An object model captures a static structure of a system by showing the objects in the system, relationship between the objects, and the attributes and operation that characterize each class of objects. Object models provide an intuitive graphic representation of a system and are valuable for communicating with customers and documenting the structure of the system.

Steps for Constructing Object Model (Class Diagram):

The first step in analyzing the requirements is to construct an object model. The object model shows the static data structure of the real world system and organizes it into workable pieces. The object model describes real world object classes and their relationships to each other.

The following steps are performed in constructing an object model:

(1) Identify objects and classes

(2) Prepare a data dictionary

(3) Identify associations (including aggregations) between objects.

(4) Identify attributes of objects and links.

(5) Organize and simplify object classes using inheritance.

(6) Identify operations to be included in a class.

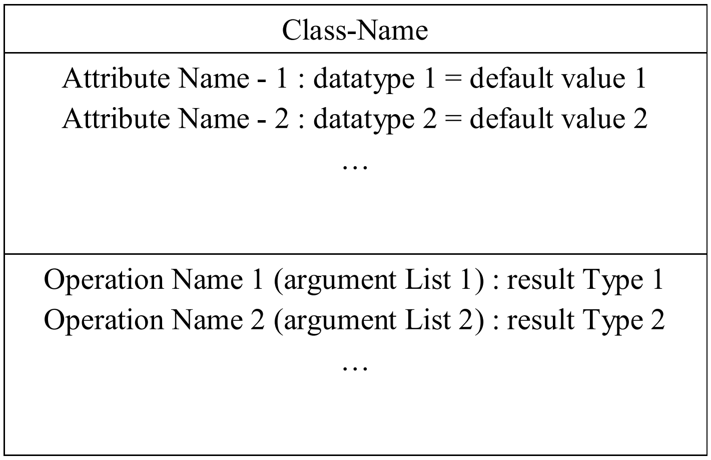
(7) Verify that access paths exist for likely queries.

(8) Iterate and refine the model

(9) Group classes into modules.

Elements of Class Diagram:

Class: Classes are composed of three things: a name, attributes, and operations. Below is an example of a class.


Class-Name
Attribute Name - 1 : datatype 1 = default value 1
Attribute Name - 2 : datatype 2 = default value 2
…
Operation Name 1 (argument List 1) : result Type 1
Operation Name 2 (argument List 2) : result Type 2
…

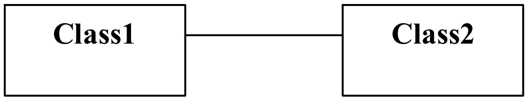
Class

Attributes: An attribute is data value held by the objects in a class. An attribute should be a pure data value, not an object. Unlike objects, pure data values do not have identity.

Operations and Methods: An operation is a function or transformation that may be applied to or by objects in a class. Operations are listed in the lower third of the class box.

Links and Association: Links and association are the means for establishing relationships among objects and classes. A link is a physical or conceptual connection between object instances. An association describes a group of links with common structure and common semantics.

An association describes a set of potential links in the same way that a class describes a set of potential objects. Associations are inherently bi-directional.



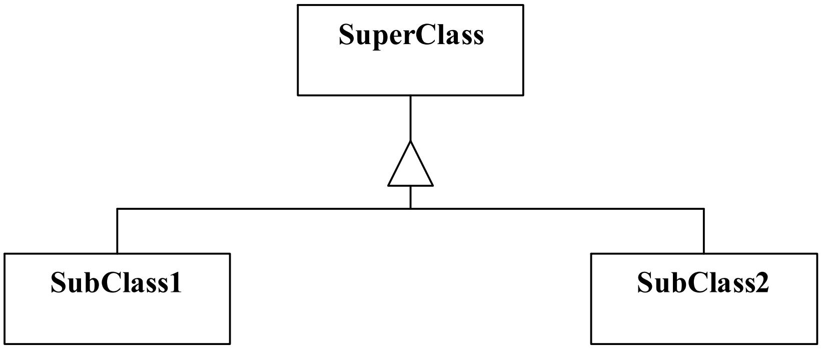
Association

Multiplicity: Multiplicity specifies how many instances of one class may relate to a single instance of an associated class. Multiplicity constrains the number of related objects. Multiplicity is often described as being “one” or “many” but more generally it is subset of non negative integers.

Object diagram indicate multiplicity with special symbols at the ends of association lines. Multiplicity can be specified with a number or set of intervals, such as “1”, “1+”(1 or more), “3-5”(3 to 5, inclusive), and “2,4,18” (2,4 or 18) .

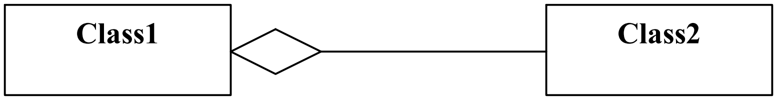
Generalization and Inheritance: Generalization and Inheritance are powerful abstractions for sharing similarities among classes while preserving their differences.

Generalization is the relationship between a class and one or more refined versions of it. The class being refined is called the superclass and each refined version is called subclass. Attributes and operations common to a group of subclasses are attached to the superclass and shared by each subclass. Each subclass is said to inherit the features of its superclass. Generalization is sometimes called the "is-a" relationship because each instance of a subclass is an instance of the superclass as well.



Generalization

Aggregation: Aggregation is the "part- whole" or "a-part-of" relationship is which objects representing the components of something are associated with an object representing the entire assembly. Aggregation is a tightly coupled form of association with some extra semantics. The most significant property of aggregation is transitivity that is if A is part of B, and B is part of C then A is part of C. Aggregation is also antisymmetric, that is if A is part of B then B is not part of A. Finally, some properties of the assembly propagate to the components as well possibly with some local modifications.



Documenting Class Diagram:

Class Id	Unique Id of Class
Class Name	Name of Class
Attributes	List of attributes for each class
Methods	Functions carried out by class
Associations	Relationship between different classes
Inheritance	Classes sharing similarities

Multiplicity	Identify how many instances of one class may relate to a single instance of an associated class
Description	Description of diagram

Conclusion: Thus class diagram is useful for abstract modeling and for designing actual programs.

Experiment No 4

Aim: To draw Use Case diagram.

Theory: The Object Oriented analysis phase of the Unified approach uses Use Case diagram to describe the system from user's perspective.

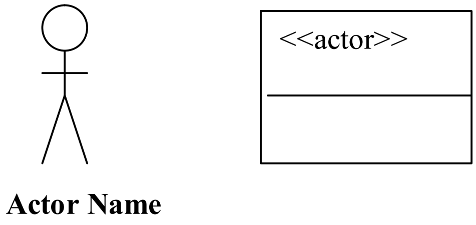
Need of Use Case:

Use Case describes the behavior of a system from a user's standpoint, Provides functional description of a system and its major processes, Provides graphic description of the users of a system and what kinds of inheritance to expect within that system, Displays the details of the processes that occur within the application area, Used to design the test cases for testing the functionality of the system.

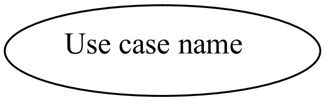
Elements of Use Case:

A Use Case diagram is quite simple in nature and depicts two types of elements: one representing the business roles and the other representing the business processes. Elements of Use Case diagrams are:

Actors: An actor portrays any entity (or entities) that perform that perform certain roles in a given system. An actor in a use case diagram interacts with a use case and is depicted "outside" the system boundary.



Use case: A use case in a Use Case diagram is a visual representation of distinct business functionality in a system. Each use case is a sequence of transactions performed by the system that produces a major suit for the actor.



System Boundary: A system boundary defines the scope of what a system will be. A system cannot have infinite functionality.

Relationships in Use Case:

1. **Include:** Include is used when two or more use cases share common portion in the flow of events. The stereotype <<include>> identifies the relationship include.

2. **Extend:** In an extend relationship between two use cases, the child use case adds to the existing functionality and characteristics of the parent use case.

3. **Generalization:** A generalization relationship is also a parent-child relationship between use cases. The child use case in the generalization relationship has the underlying business process meaning, but is an enhancement of the parent use case.

Documenting Use Case:

Use case name	Name of use case
Use case Id	Unique identifier of the use case
Super use case	If the use case inherits a parent use case then the name is to enter into this field.
Actor	The actor which are participating in the execution of use case.
Brief Description	Description of scope of use case and observable value to actor.
Preconditions	Constraints that must be satisfied before the use case can be invoked.
Post conditions	Condition will be established after the use case.
Priority	Development priority from the view of development team
Flow of events	Ste by step description of the interaction between actor and the system.
Alternative flows and exceptions	Alternatives or exceptions (other than the normal flow) that may occur.
Non-behavioral requirements	Non-functional requirements like h/w and s/w requirements.
Assumptions	All the assumptions made about the use case.
Issues	All outstanding issues related to the use case need to be resolved.
Source	Includes references and materials used in developing use case

Conclusion: Thus Use Case diagram helps to understand the different processes and entities involved in a system and help the analyst to understand and document the system.

Experiment No 5

Aim: To draw Sequence diagram.

Theory: A sequence diagram is a graphical view of a scenario that shows object interaction in a time-based sequence. Sequence diagrams are closely related to collaboration diagrams and both are alternate representations of an interaction. There are two main differences between sequence and collaboration diagrams: sequence diagrams show time-based object interaction while collaboration diagrams show how objects associate with each other.

A sequence diagram has two dimensions: typically, vertical placement represents time and horizontal placement represents different objects.

Need of Sequence Diagram:

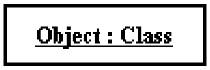
Sequence diagrams establish the roles of objects and help provide essential information to determine class responsibilities and interfaces. This type of diagram is best used during early analysis phases in design because they are simple and easy to comprehend. Sequence diagrams are normally associated with use cases.

Elements of Sequence Diagram:

The following tools located on the sequence diagram toolbox enable to model sequence diagrams:

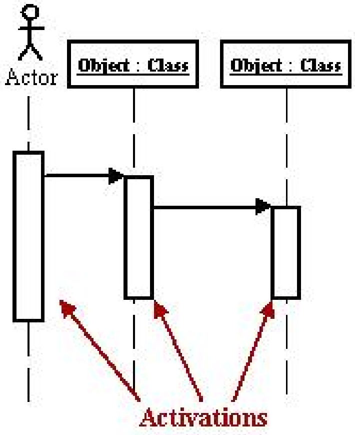
Classroles

Class roles describe the way an object will behave in context. Use the UML object symbol to illustrate class roles, but don't list object attributes.



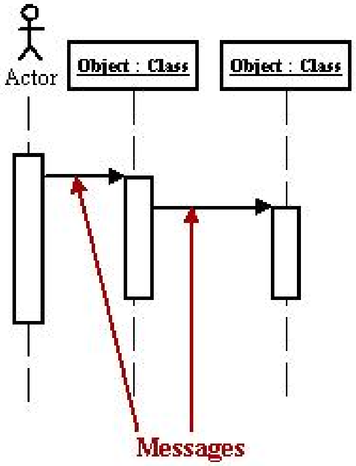
Activation

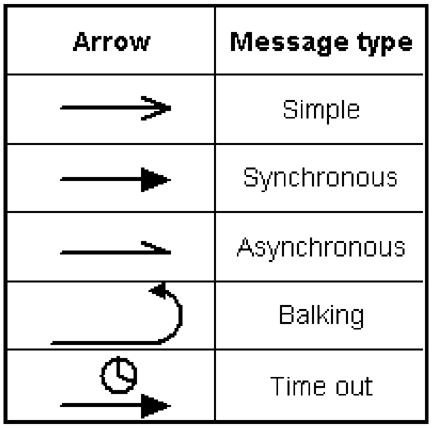
Activation boxes represent the time an object needs to comple task.



Messages

Messages are arrows that represent communication between objects. Asynchronous messages are sent from an object that will not wait for a response from the receiver before continuing its tasks.

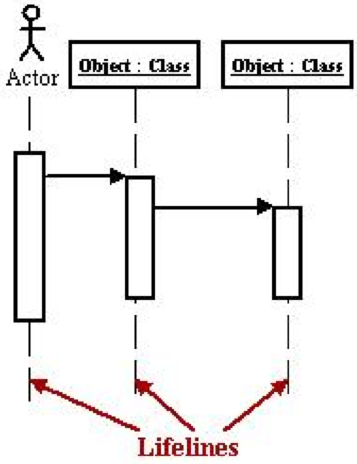




Arrow	Message type
⟶	Simple
—▶	Synchronous
⇀	Asynchronous
⤴	Balking
◷ —▶	Time out

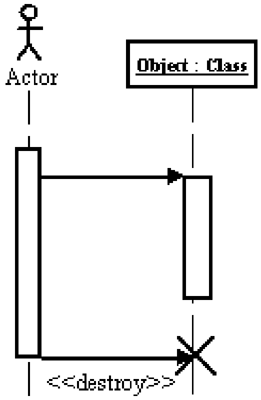
Lifelines

Lifelines are vertical dashed lines that indicate the object's presence over time.

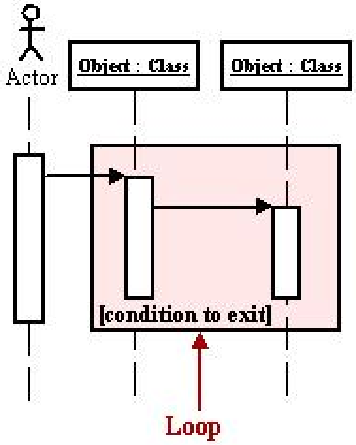


Destroying Objects

Objects can be terminated early using an arrow labeled "< < destroy > >" that points to an X.



Loops
A repetition or loop within a sequence diagram is depicted as a rectangle. Place the condition for exiting the loop at the bottom left corner in square brackets [ ].



Documenting Sequence Diagram:

Scenario Id
Scenario Name
Objects Participating in Sequence
Sequence of Operations in Scenario

Conclusion: Thus Sequence diagram helps to model different objects in a system and to represent the sequence of operations in scenario.

Experiment No 6

Aim: To draw Collaboration diagram.

Theory: A collaboration diagram is an interaction diagram that emphasizes the structural organization of the objects that send and receive messages.

We form a collaboration diagram by first placing the objects that participate in the interaction as the vertices in a graph, and then add the links that connect these objects as the arcs of this graph. Finally, adorn these links with the messages that objects sends and receive with sequence numbers.

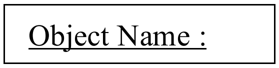
Need of Collaboration Diagram:

We use collaboration diagram to describe a specific scenario. Numbered arrows show the movement of messages during the course of a scenario. A distinguishing feature of a Collaboration diagram is that it shows the objects and their association with other objects in the system apart from how they interact with each other. The association between objects is not represented in a Sequence diagram.

Elements of Collaboration Diagram:

A sophisticated modeling tool can easily convert a collaboration diagram into a sequence diagram and the vice versa. Hence, the elements of a Collaboration diagram are essentially the same as that of a Sequence diagram.

Object: The objects interacting with each other in the system. Depicted by a rectangle with the name of the object in it, preceded by a colon and underlined.



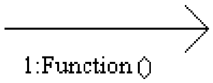
Object

Relation/Association: A link connecting the associated objects. Qualifiers can be placed on either end of the association to depict cardinality.

*   1..*

Association

Messages: An arrow pointing from the commencing object to the destination object shows the interaction between the objects. The number represents the order/sequence of this interaction.



Message

Documenting Collaboration Diagram:

Scenario Id
Scenario Name
Objects Participating in Collaboration
Association between Objects
Sequence of Operations in Scenario

Conclusion: Thus Collaboration diagram helps to model different objects in a system and to represent the associations between the objects as links.

Experiment No 7

Aim: To draw an Activity Diagram

Theory:

Activity diagram is used for business process modeling, for modeling the logic captured by a single use case or usage scenario, or for modeling the detailed logic of a business rule. Activity diagram is a dynamic diagram that shows the activity and the event that causes the object to be in the particular state. The easiest way to visualize an activity diagram is to think of a flowchart and data flow diagrams (DFDs).

Need of an Activity Diagram:

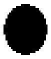
The general purpose of activity diagrams is to focus on flows driven by internal processing vs. external events.

Activity diagrams are also useful for: analyzing a use case by describing what actions needs to take place and when they should occur; describing a complicated sequential algorithm; and modeling applications with parallel processes.

Elements of an Activity Diagram

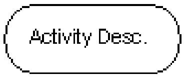
An Activity diagram consists of the following behavioral elements:

Initial Activity : This shows the starting point or first activity of the flow and denoted by a solid circle. There can only be one initial state on a diagram.



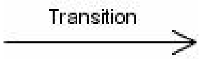
Initial Activity

Activity: Represented by a rectangle with rounded (almost oval) edges.



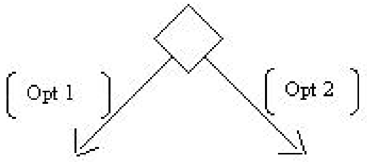
Activity

Transition: When an activity state is completed, processing moves to another activity state. Transitions are used to mark this movement. Transitions are modeled using arrows.



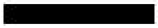
Transition

Decisions: Similar to flowcharts, a logic where a decision is to be made is depicted by a diamond, with the options written on either side of the arrows emerging from the diamond, within box brackets.



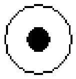
Decisions

Synchronization Bar: Activities often can be done in parallel. To split processing ("fork"), or to resume processing when multiple activities have been completed ("join"), Synchronization Bars are used. These are modeled as solid rectangles, with multiple transitions going in and/or out. Fork denotes the beginning of parallel activity. Join denotes the end of parallel processing.



Synchronization Bar

Final Activity: The end of the Activity diagram is shown by a bull's eye symbol, also called as a final activity. An activity diagram can have zero or more activity final nodes.



Final Activity

Swim Lanes

Activity diagrams provide another ability, to clarify which actor performs which activity. If you wish to distinguish in an activity diagram the activities carried out by individual actors, vertical columns are first made, separated by thick vertical black lines, termed *swim lanes* and name each of these columns with the name of the actor involved. You place each of the activities below the actor performing these activities and then show how these activities are connected.

Documenting Activity Diagram:

Activity name	Name of the activity
Description	Description about activity
Events	Events occurred during activity
Actor	Actor performing activity
Join synchronization bar	To show multiple activities occurring simultaneously
Fork synchronization bar	To resume processing when multiple activities have been completed
Activity Diagram Id	ID that identifies diagram uniquely
Name	Name o the diagram
Preconditions	Conditions before starting the activites
Post Conditions	Conditions after the activities are over

Conclusion: It can be concluded saying activity diagrams focus on flows driven by internal processing vs. external events and used to model a logic captured by single scenario.

Experiment No 8

Aim: To draw a State chart Diagram

Theory:

State chart diagrams model the dynamic behavior of individual classes or any other kind of object. They show the sequences of states that an object goes through, the events that cause a transition from one state to another and the actions that result from a state change.

State chart diagrams are closely related to activity diagrams. The main difference between the two diagrams is state chart diagrams are state centric, while activity diagrams are activity centric.

Need of State chart Diagram:

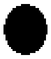
A state chart diagram is typically used to model the discrete stages of an object's lifetime, whereas an activity diagram is better suited to model the sequence of activities in a process.

Each state represents a named condition during the life of an object during which it satisfies some condition or waits for some event. A state chart diagram typically contains one start state and multiple end states. Transitions connect the various states on the diagram. As with activity diagrams, decisions, synchronizations, and activities may also appear on state chart diagrams.

Elements of State chart Diagram

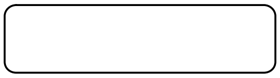
A State chart diagram consists of the following behavioral elements:

Start State : The start state icon is a small, filled circle that may contain a name (Begin Process):



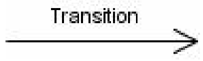
Initial State

State: The state icon appears as a rectangle with rounded corners and a name. The name of a state should be unique to its enclosing class, or if nested, within the state. All state icons with the same name in a given diagram represent the same state.



State

Transition: When an activity state is completed, processing moves to another activity state. Transitions are used to mark this movement. Transitions are modeled using arrows.



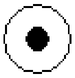
Transition

Actions : Actions on states can occur at one of four times:

- on entry
- on exit
- do
- on event.

An on event action is similar to a state transition label with the following syntax:
event(args)[condition] : the Action

End State: The end state icon is a filled circle inside a slightly larger unfilled circle that may contain a name (End Process):



End State

Documenting State Chart Diagram:

State Name	Task performed by that object
State Id	Unique id of that state
Activity	Activity performed.
Transitions	Identify transitions to move from one activity to another activity

Conclusion: It can be concluded saying state chart diagrams model the discrete stages of an object’s lifetime.

Experiment No 9

Aim: To draw Component diagram.

Theory:

A component represents a software module with a well-defined interface. The interface of a component is represented by one or several interface elements that the component provides. Components are used to show compiler and run-time dependencies, as well as interface and calling dependencies among software modules. They also show which components implement a specific class.

A system may be composed of several software modules of different kinds. Each software module is represented by a component in the model. To distinguish different kinds of components from each other, stereotypes are used.

Need of Component Diagram:

We use component diagrams to visualize the static aspect of the physical components and their relationships and to specify their details for construction; this involves modeling the physical things that reside on a node, such as executables, libraries, tables, files, and documents.

When we model the static implementation view of a system, we will typically use component diagrams in one of four ways.

(1) ***To model source code :*** With most contemporary object-oriented programming languages, we will cut code using integrated development environments that store our source code in files. We can use component diagrams to model the configuration management of these files, which represent work-product components.

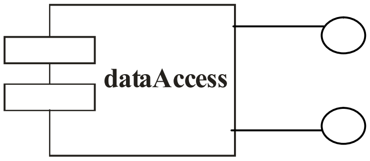
(2) ***To model executable releases :*** A release is a relatively complete and consistent set of artifacts delivered to an internal or external user. In the context of components, a release focuses on the parts necessary to deliver a running system. When we model a release using component diagrams, are visualizing, specifying, and documenting the decisions about the physical parts that constitute our software-that is, its deployment components.

(3) ***To model physical databases :*** Think of a physical database as the concrete realization of a schema, living in the world of bits. Schemas, in effect, offer an API to persistent information; the model of a physical database represents the storage of that information in the tables of a relational database or the pages of an object-oriented database. We use component diagrams to represent these and other kinds of physical databases.

(4) ***To model adaptable systems :*** Some systems are quite static; their components enter the scene, participate in an execution, and then depart. Other systems are more dynamic, involving mobile agents or components that migrate for purposes of load balancing and failure recovery. We use component diagrams in conjunction with some of the UML's diagrams for modeling behavior to represent these kinds of systems.

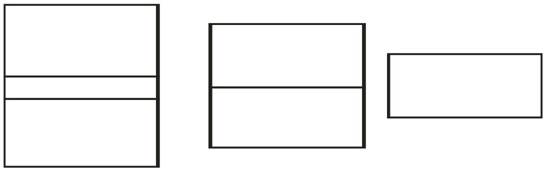
Elements of Component Diagram:

Component: A component is a physical and replaceable part of a system that conforms to and provides the realization of a set of interfaces. Graphically, a component is rendered as a rectangle with tabs, with the name of the object in it, preceded by a colon and underlined.



Component

Class/Interface/Object: Similar to the notations used in class and object diagrams



Class/Interface/Object notations

Relation/Association: Similar to the relation/association used in class diagrams

Relations/Associations

Like all other diagrams, component diagrams may contain notes and constraints.

Component diagrams may also contain packages or subsystems, both of which are used to group elements of our model into larger chunks. Sometimes, we want to place instances in our component diagrams, as well, especially when we want to visualize one instance of a family of component-based systems.

Documenting Component Diagram:

Component Id

Component Name

Dependencies between Components

Component Description

Conclusion: Thus Component diagram helps to understand the structure of code and to represent the different high-level reusable parts of a system.

Experiment No 10

Aim: To draw Deployment diagram.

Theory: *A deployment diagram* is *a diagram that shows the configuration of run time processing nodes and the components that live on them*. This diagram is by far more useful when a system is built and ready to be deployed. But, this does not mean that we should start on our deployment diagram after our system is built. On the contrary, our deployment diagram should start from the time our static design is being formalized using, say, class diagrams. This deployment diagram then evolves and is revised until the system is built. It is always a best practice to have visibility of what our deployment environment is going to be before the system is built so that any deployment-related issues are identified to be resolved and not crop up at the last minute.

Need of Deployment Diagram:

We use deployment diagrams to model the static deployment view of a system. For the most part, this involves modeling the topology of the hardware on which our system executes. Deployment diagrams are essentially class diagrams that focus on a system's nodes.

When we model the static deployment view of a system, we will typically use deployment diagrams in one of three ways.

1. ***To model embedded systems***

An embedded system is *a software-intensive collection of hardware that inter. faces with the physical world*. Embedded systems involve software that controls devices such as motors, actuators, and displays and that, in turn, is controlled by external stimuli such as sensor input, movement, and temperature changes. We can use deployment diagrams to model the devices and processors that comprise an embedded system.

2. ***To model client/server systems***

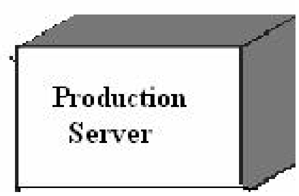
A client/server system is *a common architecture focused on making a clear separation of concerns between the system's user interface (which lives on the client) and the system's persistent data (which lives on the server)*. Client server systems are one end of the continuum of distributed systems and require you to make decisions about the network connectivity of clients to servers and about the physical distribution of your system's software components across the nodes. We can model the topology of such systems by using deployment diagrams.

3. ***To model fully distributed systems***

At the other end of the continuum of distributed systems are those that are widely, if not globally, distributed: typically encompassing multiple levels of servers. Such systems are often hosts to multiple versions of software components, some of which may even migrate from node to node. Crafting such systems requires we to make decisions that enable the continuous change in the system's topology. we can use deployment diagrams to visualize the system's current topology and distribution of components to reason about the impact of changes on that topology.

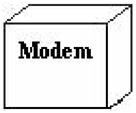
Elements of Deployment Diagram:

Processor: Processor is a piece of hardware capable of executing programs. The processor is represented as shaded cube with a name of object. A processor can have list of processes on it.



Processor

Device: A piece of hardware incapable of executing program is called as device. Device will also have on a cube.



Device

Connection : Similar to the relation/association used in class diagrams to define the interconnection between nodes.

Connections

Like all other diagrams, deployment diagrams may contain notes and constraints.

Deployment diagrams may also contain components, each of which must live on some node. Deployment diagrams may also contain packages or subsystems, both of which are used to group elements of our model into larger chunks. Sometimes, we will want to place instances in your deployment diagrams, as well, especially when we want to visualize one instance of a family of hardware topologies.

Note : In many ways, a deployment diagram is just a special kind of class diagram, which focuses on a system's nodes.

Documenting Deployment Diagram:

Machine Id	Unique id of machine on which component is placed
Machine Name	Name of machine on which component is placed
Node Type	Identify type of node i.e., Processor or Device used.
Machine Location	Location of machine
Components	List of components used in diagram
Description	Detail description of diagram

Conclusion: Thus Deployment diagram provides a different perspective of the application and captures the configuration of the runtime elements of the application.

Experiment No 11

Aim: To design Test Cases for the system.

Theory:

The Classical strategy for testing computer software begins with “testing in the small” and work outward toward “testing in large”. We begin with unit testing, then progress toward integration testing, and culminate with validation and system testing. In conventional applications, unit testing focuses on the smallest compliable program unit. Once each of these units has been tested individually, it is integrated into program structure while a series of regression tests are run to uncover errors due to interfacing between modules and side effects caused by addition of new units. Finally, the system as a whole is tested to ensure that errors in requirements are uncovered.

Unit Testing in the OO Context

When OO software is considered, the concept of the unit change. Encapsulation drives the definition of classes and objects. This means that each class instance of a class packages attributes and the operations that manipulate these data. Rather than testing an individual module, the smallest testable unit is the encapsulated class or object. Because class can contain a number of different operations and a particular operation may exist as a part of a number different classes, the meaning of unit testing changes dramatically.

Integration Testing in the OO Context

There are two different strategies for integration testing of OO systems.

Thread Based Testing: Integrates the set of classes required to respond to one input or event for the system. Each thread is integrated and tested individually. Regression testing is applied to ensure that no side effects occur.

Use-Based Testing: This approach begins with the construction of the system by testing those classes (called independent classes) that use very few of server classes. After independent classes are tested, the next layers of classes, called dependent classes, that use the independent classes are tested. This sequence of testing layers of dependent classes continues until the entire system is constructed.

Cluster Testing is one step in the integration testing of OO software. Here, a cluster of collaborating classes (determined by examining the CRC and object-relationship model) is exercised by designing test cases that attempt to uncover errors in the collaborations.

Validation Testing in an OO Context:

At the validation or system level, the details of class connections disappear. Like conventional validation, the validation of OO software focuses on user visible actions and user recognizable output from the system. To assist in the derivation of validation tests, the tester should draw upon the use-cases that are a part of the analysis model. The use-case provides a scenario that has a high likelihood of uncovered errors in user interaction requirements.

Conventional black-box testing methods can be used to drive validation tests. In addition, test cases may be derived from the object-behavior model and from the event flow diagram created as a part of OOA.

Documenting Test Cases:

Test Case Name	**Input**	**Expected Output**	**Actual Output**

Conclusion: Thus different test cases were designed that has a high probability of finding an as yet undiscovered error.